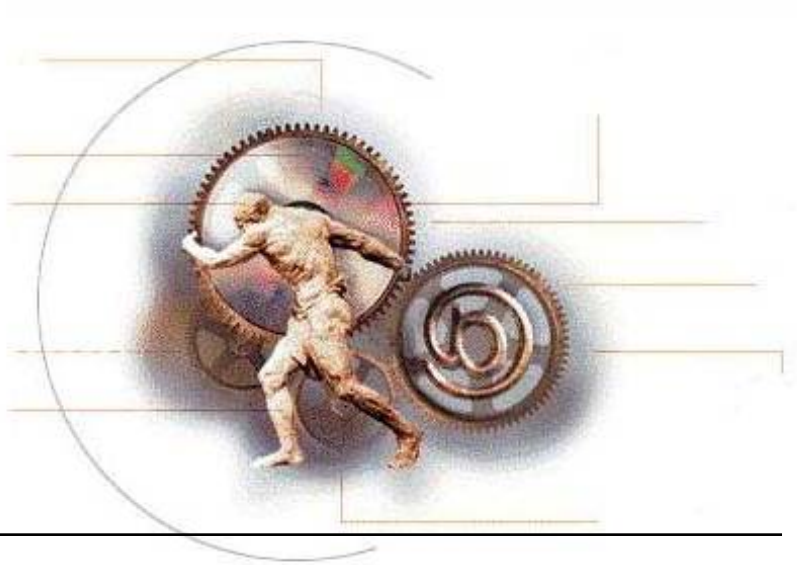


Course 5

切割與征服
Divide-and-Conquer

國立聯合大學 資訊管理學系 陳士杰老師

Outlines

- 本章重點
  - Divide-and-Conquer策略的描述
  - Binary Search
  - Merge Sort
  - Divide-and-Conquer 的技巧
  - Quick Sort
  - Strassen's 矩陣相乘演算法
  - 何時不能使用 Divide-and-Conquer

Divide-and-Conquer策略的描述與技巧

- Divide-and-conquer 是一種由上而下 (top-down) 的解題方式.
  - 它將一個問題切割 (divides) 成兩個或以上的較小問題。較小的問題通常是原問題的實例。
  - 如果較小的問題之解可以容易地獲得，那麼原問題的解可以藉由合併較小問題的答案獲得。
  - 如果小問題還是太大以致於不易解決，則可以再被切割成更小的問題直到切到夠小而易獲得結果爲止。

- **Def:**
  - 可將母問題切割成**較小的問題** (切割)，使用**相同的解決程序**加以處理 (征服)。所有小問題的解可以成爲母問題的最後解; 若有必要，則再將每個小問題的處理結果加以合併，就可以得到最後的答案。
    - 由於使用相同的解決程序處理每個小問題，這一個程序就會被遞迴呼叫，因此一個遞迴演算法則通常以一個副程式的型式出現，內部包含一個解決程序與遞迴呼叫。
    - 對於具有**遞迴關係**的問題，或是一些採用遞迴定義的資料結構，都適合採用**Divide-and-Conquer**演算法設計策略
  - 最簡潔、易懂
  - 效率差 (∵採用遞迴設計)

Divide-and-Conquer使用時機

- 下列兩種情況是適合使用Divide-and-Conquer設計策略 (也是遞迴演算法的適用時機):
  - 問題本身具有遞迴關係
    - 母問題可被切割成較小的 "相同" 問題
    - 如: 階乘問題、費氏數問題、河內塔問題、快速排序問題、二元搜尋問題...等
  - 資料結構屬於遞迴定義
    - 大量的Data Set，在切割後仍爲一組具 "相同性質" 的Data Set
    - 如: 二元樹 (Binary Tree)、鏈結串列 (Link List)...等

遞迴演算法則的設計

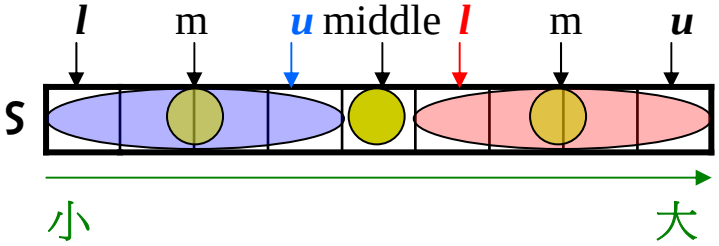
1. 找出問題的終止條件.
2. 找出問題本身的遞迴關係 (遞迴呼叫).

- 技巧:
  - 思考遞迴呼叫需要哪些參數?
  - 遞迴呼叫的傳回值爲何?
  - 遞迴呼叫的終止條件爲何? 終止傳回何值?

```
Procedure Recursion_subroutine(Parameter);
{
  if (終止條件) then Return( );
  else Recursion_subroutine(New_parameter) ;
}
```

Binary Search (二分搜尋)

- 實施前提:
  - 檔案中記錄須事先由小到大排序過
  - 須由**Random (或Direct) access**之機制支援 (e.g., Array)
- 觀念:
  - 每次皆與Search範圍的中間記錄進行比較!!



$$middle = \left\lfloor \frac{l+u}{2} \right\rfloor$$

while ($l \le u$)

$$m = \left\lfloor \frac{l+u}{2} \right\rfloor$$

比較 (k, S[m])

```
case "=": found, i = m, return i;   //找到了
case "<": u = m-1;                  //要找的資料在左半部
case ">": l = m+1;                  //要找的資料在右半部
recurn 0;
```

分析

- 利用Time function

$T(n) = T(n/2) + O(1)$

$= T(n/2) + c$

$= (T(n/4 + c)) + c = T(n/4) + 2c$

$= (T(n/8) + c) + 2c = T(n/8) + 3c$

$= \ldots$

$= T(n/n) + \log_2 n \times c$

$= T(1) + c \log_2 n$ (T(1) = 1, c 爲大於 0 的常數)

$= 1 + c \log_2 n$

$\therefore T(n) = O(\log_2 n)$

- 二元搜尋法的步驟摘要如下。如果x與中間項相同則離開，否則:
  - 切割 (Divide) 該陣列成大約一半大小的兩個子陣列。.
    - 如果x小於中間項，選擇左邊的子陣列。如果x大於中間項，則選擇右邊的子陣列。
  - 藉由判斷x是否在該子陣列中來征服 (Conquer; 或稱解決 solve) 該子陣列。
    - 除非該子陣列夠小，否則使用遞迴來做這件事。
  - 由子陣列的解答來獲得 (Obtain) 該陣列的解答。
- 二元搜尋法是最簡單的一種Divide-and-conquer演算法。因爲原有問題的解答就是較小問題所解出的解答，所以沒有輸出結果的合併。

Merge Sort (合併排序)

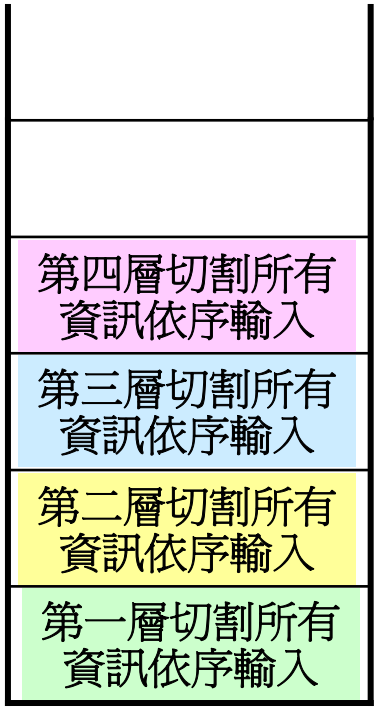
- 觀念:
  - 將兩個已排序過的記錄合併，而得到另一個排序好的記錄。
- 可分爲兩種類型:
  - **Recursive (遞迴)**
  - **Iterative (迴圈, 非遞迴)**

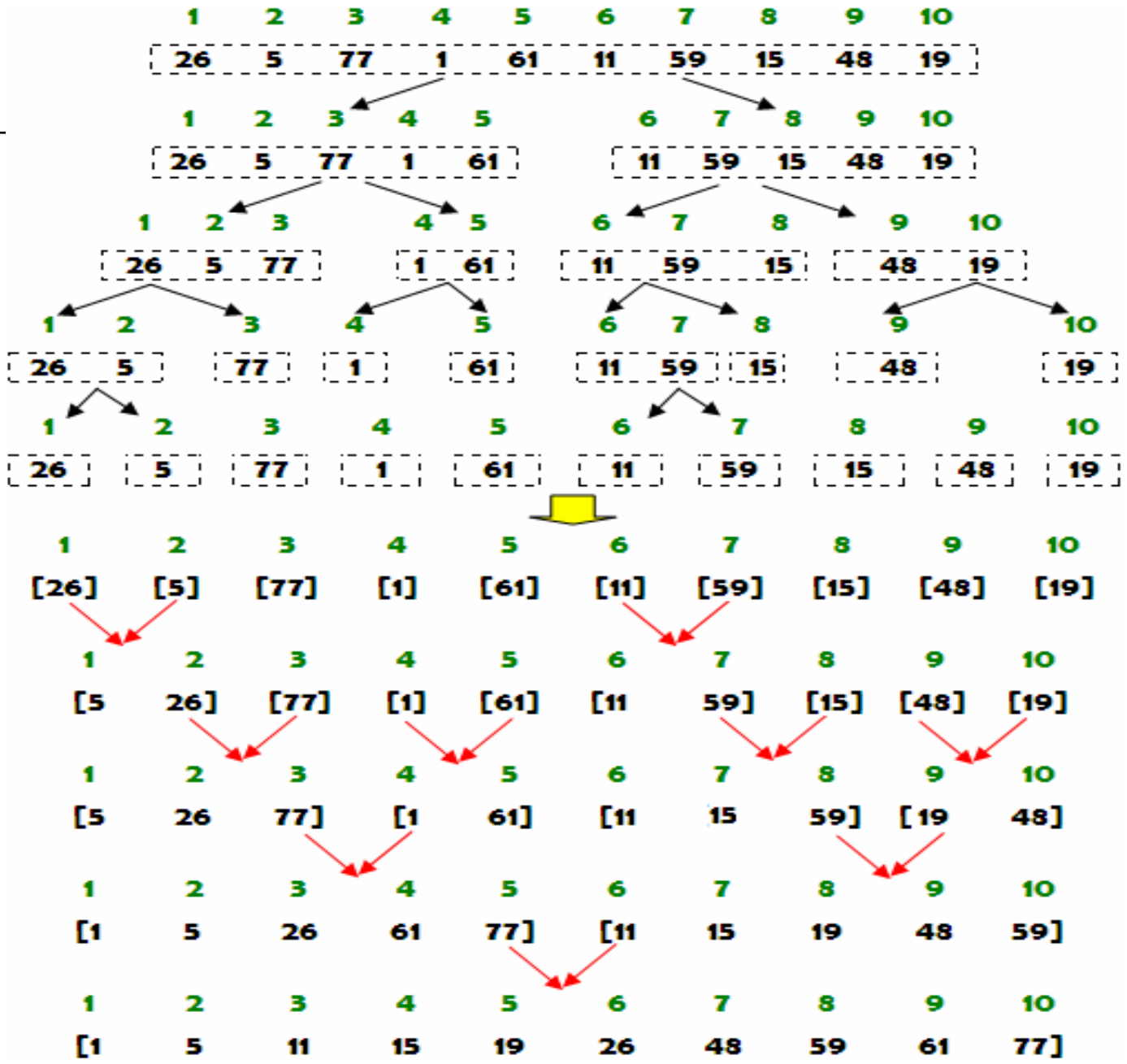
Recursive Merge Sort (遞迴合併排序)

- 將資料量 n 切成 n/2 與 n/2 兩半部，再各自Merge Sort，最後合併兩半部之排序結果即成。

- 切割資料量 n 的公式爲: $\left\lfloor \frac{(low + high)}{2} \right\rfloor$

- [ ]: Run, 已排序好的檔案記錄
- Run的長度: Run中記錄個數


第四層切割所有資訊依序輸入
第三層切割所有資訊依序輸入
第二層切割所有資訊依序輸入
第一層切割所有資訊依序輸入
Stack


1 2 3 4 5 6 7 8 9 10
26 5 77 1 61 11 59 15 48 19
1 2 3 4 5
26 5 77 1 61
6 7 8 9 10
11 59 15 48 19
1 2 3
26 5 77
4 5
1 61
6 7 8
11 59 15
9 10
48 19
1 2
26 5
3
77
4 5
1 61
6 7 8
11 59 15
9 10
48 19
1 2 3 4 5 6 7 8 9 10
26 5 77 1 61 11 59 15 48 19
1 2 3 4 5 6 7 8 9 10
[26] [5] [77] [1] [61] [11] [59] [15] [48] [19]
1 2 3 4 5 6 7 8 9 10
[5 26] [77] [1] [61] [11 59] [15] [48] [19]
1 2 3 4 5 6 7 8 9 10
[5 26 77] [1 61] [11 15 59] [19 48]
1 2 3 4 5 6 7 8 9 10
[1 5 26 61 77] [11 15 19 48 59]
1 2 3 4 5 6 7 8 9 10
[1 5 11 15 19 26 48 59 61 77]

Time-Complexity

- Avg. / Worst / Best Case: **O(n log n)**
- 以Recursive Merge Sort角度:

[說明]: 左半部遞迴 右半部遞迴

時間函數: T(n) = T(n/2) + T(n/2) + c×n

時間複雜度求法:

- 遞迴樹
  - 步驟:
    - 將原本問題照遞迴定義展開
    - 計算每一層的Cost
    - 加總每一層的Cost即爲所求
- 數學解法

- 最後合併左右兩半部所花時間
  - ∵ 左、右半部排好之後，各只剩一個Run，且兩半部各有n/2的資料量，其最後一次合併時的比較次數"最多"爲 n/2 + n/2 -1 次，即約 n-1 次 (slide 72)
  - ∴時間的表示可爲 c×n 次(∵爲線性時間))

- 合併排序法包含了下列的步驟：
  - 切割 (Divide) 該陣列成爲兩個具有 $n/2$ 個項目的子陣列。
  - 征服 (Conquer; 或稱解決solve)每一個子陣列。
    - 除非該子陣列夠小，否則使用遞迴來做這件事。
  - 合併 (Combine) 所有子陣列的所有解答，以獲得主陣列的解答。

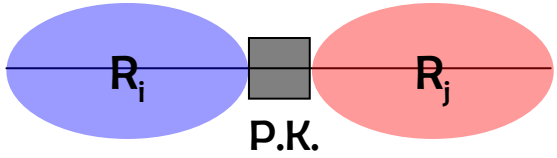
Divide-and-Conquer 技巧

- Divide-and-conquer 的設計策略包含下列的步驟:
  - 切割 (Divide) 一個較大的問題以成爲一個或多個較小的問題。
  - 征服 (Conquer ; 或稱解決solve) 每一個較小的問題。
    - 除非問題已經足夠的小，否則使用遞迴來解決。
  - 如果需要, 將所有小問題的解答加以合併(combine) ，以獲得原始問題的解答。
    - 需要合併的問題: Merge sort
    - 不需要合併的問題: Binary search

Quick Sort (快速排序)

- Avg. case 下，排序最快的algo.
- Def:
  - 將大且複雜的問題切成許多獨立的小問題，再加以解決各小問題後，即可求出問題的Solution。
  - 此即 "**Divide-and-Conquer**" (切割並征服)的解題策略。
- 觀念:
  - 將第一筆記錄視爲Pivot Key (樞紐鍵 (P.K.) ，或稱Control Key)，在Pass 1 (第一回合) 後，可將P.K.置於 "最正確" 的位置上。
  - Ex:

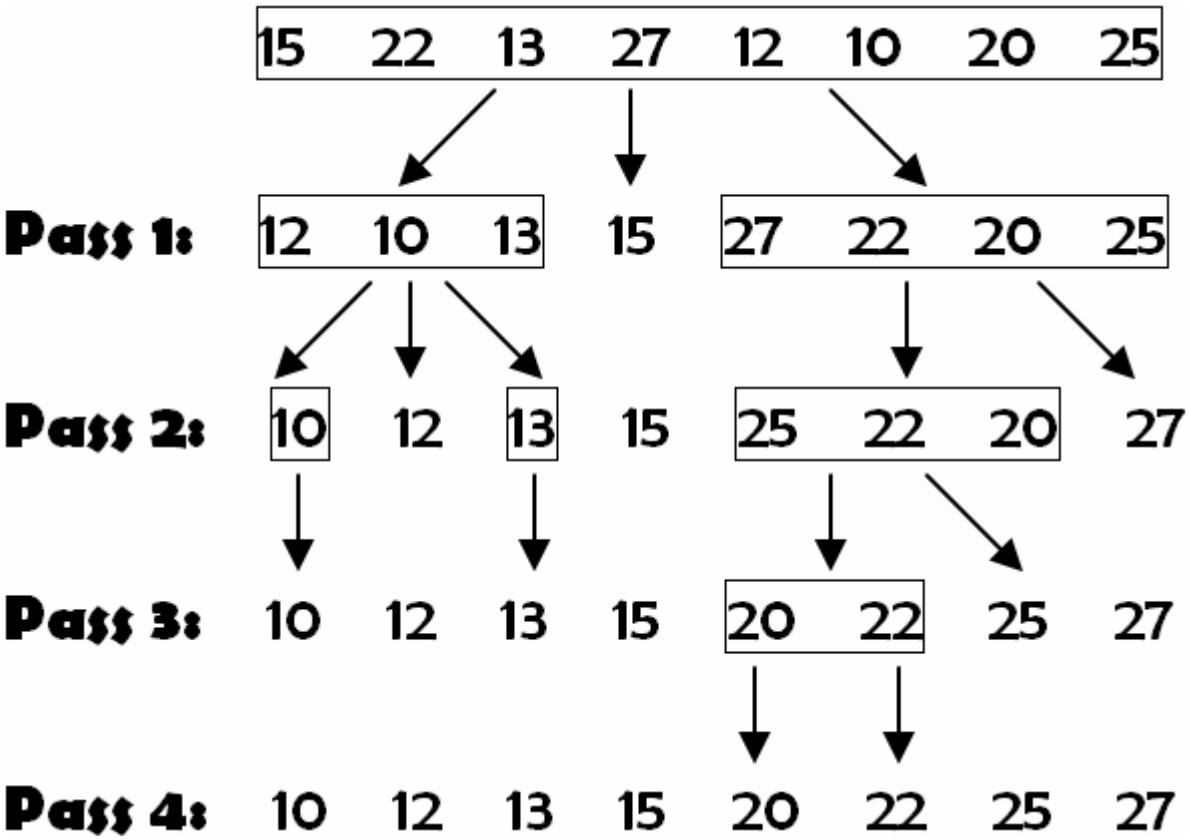
(經過Pass 1) ⇨



, $R_i.key \le P.K.$ 且 $R_j.key \ge P.K.$

  - 把P.K.擺在正確的位置 ⇨爲切割的概念 (∵可使用遞迴)

- 多顆CPU時的運算過程:



Time-Complexity

- **Best Case: O(n log n)**
  - P.K.之最正確位置恰好將資料量均分成二等份
    - 以Multiprocessor來看，2個CPU的工作量相等，工作可同時做完，沒有誰等誰的問題

[說明]:

左半部 右半部

時間函數: $T(n) = c \times n + T(n/2) + T(n/2)$

時間複雜度求法:

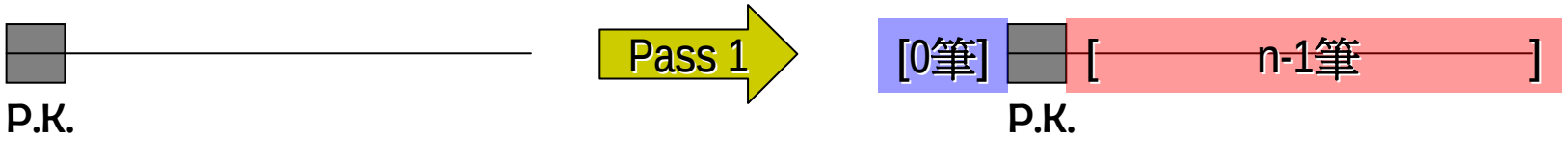
變數 i 與 j 最多花 n 個執行時間找記錄 (即: 決定P.K.最正確位置所花時間)

- 遞迴樹
  - 步驟:
    - 將原本問題照遞迴定義展開
    - 計算每一層的Cost
    - 加總每一層的Cost即爲所求
- 數學解法

- **Worst Case: $O(n^2)$**
  - 當輸入資料是由大到小或由小到大排好時 (切割毫無用處)

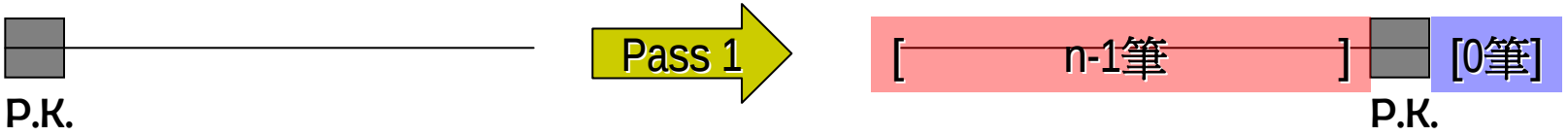
[說明]:

輸入資料: 小 大



或

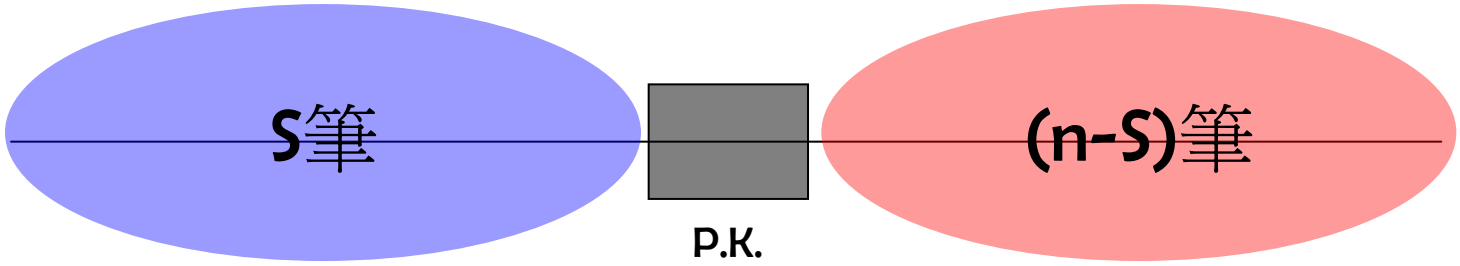
輸入資料: 大 小



- Average Case: **O(n log n)**

[說明]:

$$\Rightarrow \quad T(n) = \frac{1}{n}\sum_{s=1}^{n}[T(s) + T(n-s)] + cn \qquad , T(0) = 0$$



Strassen's Matrix Multiplication Algorithm

- 矩陣乘法問題 (Matrix Multiplication Problem):
  - 給定兩個方陣A, B，其Size均爲n×n，其中$n=2^k$。如果n不是2的冪次方，則可以增加額外的列與行，但是補上的元素都是零:
    - 若矩陣是扁的，則可以在該矩陣下方補上數列的0，使之成爲方陣
    - 若矩陣是窄的，則可以在該矩陣右方補上數行的0，使之成爲方陣
- 欲求C = A × B，傳統的矩陣乘法:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21}$

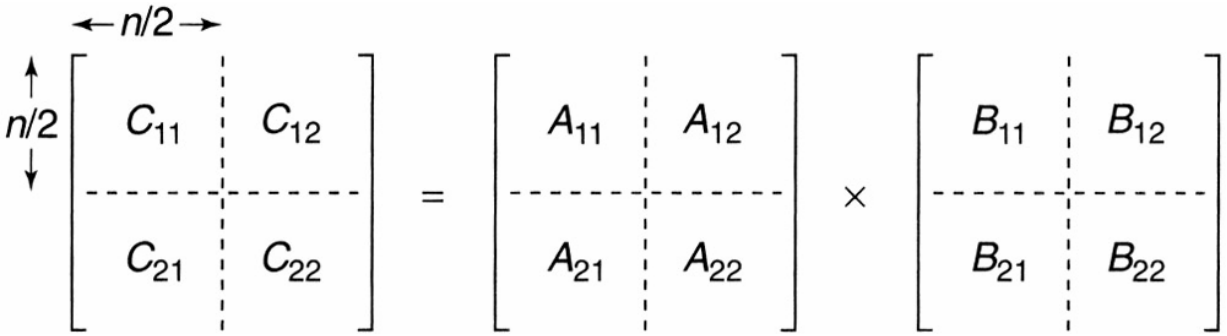
$c_{12} = a_{11} \times b_{12} + a_{12} \times b_{22}$

$c_{21} = a_{21} \times b_{11} + a_{22} \times b_{21}$

$c_{22} = a_{21} \times b_{12} + a_{22} \times b_{22}$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad \text{for } 1 \le i, j \le n.$$

- 將矩陣乘法問題放大來看:



$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$

$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$

$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$

$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$

C_{ij}, A_{ij}, B_{ij}皆爲子矩陣，即可用遞迴切割的方式來將此矩陣切割成數個小矩陣。

- 遞迴方程式爲: $T(n) = 8T(n/2) + cn^2$
- 由支配理論可以得知該遞迴方程式最後可以得到$\theta(n^3)$

Algorithm 1.4: Matrix Multiplication

Problem: Determine the product of two $n \times n$ matrices.

Inputs: a positive integer n, two-dimensional arrays of numbers A and B, each of which has both its rows and columns indexed from I to n.

Outputs: a two-dimensional array of numbers C, which has both its rows and columns indexed from 1 to n, containing the product of A and B.

```
void matrixmult (int n,
                 const number A[][],
                 const number B[][],
                 number C[][])
{
   index i, j, k;

   for (i=1; j<= n; j++)
      for (j=1; j<= n; j++){
          C[ i ] [ j ] = 0;
          for (k=1; k<= n; k++)

              C[ i ] [ j ] = C[ i ] [ j ] + A [ i ] [ k ] * B [ k ] [ j ];
       }
}
```

- 該演算法的時間複雜度爲**O(n³)**，乘法運算比加法運算要來得多。
  - 前例的乘法有**8**個，加法有**4**個。
- 然而，就系統執行的角度來說，乘法運算的複雜度遠超過加法運算，因此該演算法在實際執行的速度會更慢。

- 在1969年, Strassen 發表了一個時間複雜度較三次方演算法爲佳 (time complexity is better than cubic)的演算法。
- Strassen的方法需要 7 次乘法和 18 次的加/減法

Example 2.4

Suppose we want the product C of two 2 x 2 matrices, A and B. That is,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Strassen determined that if we let

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22})\, b_{11} \\ m_3 &= a_{11}\,(b_{12} - b_{22}) \\ m_4 &= a_{22}\,(b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12})\, b_{22} \\ m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}), \end{aligned}$$

the product C is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}.$$

- 遞迴方程式爲: $T(n) = 7T(n/2) + cn^2$
- 由支配理論可以得知該遞迴方程式最後可以得到$\theta(n^{lg7})$

Algorithm 2.8: Strassen

Problem: Determine the product of two n x n matrices where n is a power of 2.

Inputs: an integer n that is a power of 2, and two n x n matrices A and B.

Outputs: the product C of A and B.

```
void strassen (int n
                      n × n_matrix A,
                      n × n_matrix B,
                      n × n_matrix& C)
{
   if (n<=threshold)
      compute C = A × B using the standard algorithm;
   else{
      partition A into four submatrices A11, A12, A21, A22;
      partition B into four submatrices B11, B12, B21, B22;
      compute C = A × B using Strassen's method;
         // example recursive call;
         // strassen (n/2, A11 + A22, B11 + B22, M1)
  }
}
```

何時不能使用 Divide-and-Conquer

- 在下列兩種情況，我們應免使用 Divide-and-conquer:
  1) 一個大小爲n的個體被分成兩個或更多個大小爲接近n的個體.
  2) 一個大小爲n的個體被分成n個大小爲n/c的個體，其中c爲常數.

- 有時，某些問題隨輸入範例的大小成指數成長是無法避免的。雖然此時Divide-and-conquer無法獲致良好的執行效率，但仍可採用。
  - 河內塔問題每呼叫一次就需搬動圓盤一次，當圓盤的個數 n 是64時，總共需要搬動圓盤 $2^{64}-1$次，因此演算法的複雜度等級 (order) 是 $O(2^n)$
    - 即: 河內塔問題的圓盤搬動次序是與 n 成**指數關係**
  - 但是經過証明上述河內塔問題的演算法，已經是給定該問題的限制下最佳的演算法則了。